Constructing a Tree from Homeomorphic Subtrees, with Applications to Computational Evolutionary Biology

Monika Rauch Henzinger *
Valerie King †
and Tandy Warnow ‡

Abstract

We are given a set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of rooted binary trees, each T_i leaf-labeled by a subset $\mathcal{L}(T_i) \subset \{1, 2, \ldots, n\}$. If T is a tree on $\{1, 2, \ldots, n\}$, we let $T|\mathcal{L}$ denote the minimal subtree of T induced by the nodes of $\mathcal{L}$ and all their ancestors. The *consensus tree problem* asks whether there exists a tree T^* such that for every i, $T^*|\mathcal{L}(T_i)$ is homeomorphic to T_i.

We present algorithms which test if a given set of trees has a consensus tree and if so, construct one. The deterministic algorithm takes time $\min\{O(Nn^{1/2}), O(N + n^2 \log n)\}$, where $N = \sum_i |T_i|$, and uses linear space. The randomized algorithm takes time $O(N \log^3 n)$ and uses linear space. The previous best for this problem was an 1981 $O(Nn)$ algorithm by Aho et al. Our faster deterministic algorithm uses a new efficient algorithm for the following interesting dynamic graph problem: Given a graph G with n nodes and m edges and a sequence of b batches of one or more edge deletions, then after each batch, either find a new component that has just been created or determine that there is no such component. For this problem, we have a simple algorithm with running time $O(n^2 \log n + b_0 \min\{n^2, m \log n\})$, where b_0 is the number of batches which do not result in a new component. For our particular application, $b_0 \leq 1$. If all edges are deleted, then the best previously known deterministic algorithm requires time $O(m\sqrt{n})$ to solve this problem. We will also present two applications of these consensus tree algorithms which solve other problems in computational evolutionary biology.
Key words: Algorithms, Data Structures, Evolutionary Biology, and Theory of Databases.

1 Introduction

We are interested in the following problem which arises in computational biology and the theory of relational data bases.

Rooted subtree consistency:

Input: A set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ of rooted binary trees, each T_i leaf-labelled by a subset $\mathcal{L}(T_i) \subset \{1, 2, \ldots, n\}$.

*Digital Equipment Corporation, Systems Research Center, Palo Alto, CA 94301. This work was done in part which visiting at the International Computer Science Institute, Berkeley, CA. This research was supported by the NSF CAREER award CCR-9501712. Email: monika@pa.dec.com.

†Department of Computer Science, University of Victoria, Victoria, BC. Email: val@csr.uvic.ca. This research was supported by an NSERC Grant.

‡Department of Computer and Information Science, University of Pennsylvania Philadelphia, PA. Email: tandy@central.cis.upenn.edu. This work was supported in part by a National Young Investigator Award from NSF, CCR 9457800, by a fellowship from the David and Lucille Packard Foundation, the Penn Research Foundation, and by generous support from Paul Angello.

Question: Does there exist a tree T^* such that for every i, $T^*|\mathcal{L}(T_i)$ is homeomorphic to T_i, where $T^*|\mathcal{L}(T_i)$ denotes the subtree of T^* induced by the leaves of $\mathcal{L}(T_i)$ and their ancestors?

When the tree T^* exists, the set $\mathcal{T}$ is said to be a *compatible* set of subtrees, and the set T^* is called the *consensus tree.*

Our deterministic algorithm runs in time $\min\{O(Nn^{1/2}), O(N + n^2 \log n)\}$, where $N = \sum_i |T_i|$. If the randomized fully dynamic connectivity data structure of [20] is used, the resulting algorithm takes time $O(N \log^3 n)$. By contrast, the best previous algorithm for this problem was given in the 1981 paper by Aho et al [3], and had running time $O(Nn)$.

The efficiency of our algorithm results in part from an efficient solution to a variation of the following dynamic graph problem, called the *batch deletion problem*: Given a graph G with n nodes and m edges and a sequence of b batches of one or more edge deletions, then after each batch, either find a newly created component, i.e. one that has just been created, and output all its nodes, or determine that there is no such component. No information about the batches of deletions is given in advance of their occurrence.

For this problem, we have a simple algorithm with running time $O(n^2 \log n + b_0 \min(n^2, m \log n))$, where b_0 is the number of batches which do not result in a new component. For our particular application, $b_0 \leq 1$. Also, for our application, it suffices to find a previously undiscovered component, which may have been created by a previous batch of deletions. We define these two versions of the problem more formally below.

Note that the previous deterministic algorithm for this problem requires time $\Omega(d\sqrt{n})$, where $d \geq b_0$ is the total number of deletions. Hence if $d \geq n^{3/2} \log n + \min(n^{3/2}, m \log n/\sqrt{n})$, then the presented algorithm is more efficient than the best deterministic dynamic connectivity algorithms [12, 18].

The rest of the paper is organized as follows. We present our algorithms for the batch problem in Section 2, and our algorithms for testing the consistency of rooted binary subtrees in Section 3. We show how to use these algorithms for constructing evolutionary trees from biomolecular data, such as DNA, RNA, or amino-acid sequences, in Section 4.

2 The batch deletion problem

The batch deletion problem is as follows. We are given a graph G on n nodes and m edges, and an (unknown) sequence of batches of edge deletions. After each batch we wish to either determine that the number of connected components has not changed, or to find a newly created component and output the set of nodes. We may wish either to find all such components, or simply just one. (We will say a component has been "found" or "discovered" if its nodes and no other nodes in the graph have the same label.) Let b_0 be the number of batches which do not result in a new component.

The batch deletion problem can be solved by computing connectivity from scratch after each batch. If there were b batches of deletions, this would result in bm running time, regardless of the value of b_0.

A 1981 algorithm by Even and Shiloach [14] considered the dynamic graph problem for individual deletions only. Over the course of deleting every edge from a graph, their algorithm spends $O(m \log n)$ time processing individual deletions each of which results in a new component and $O(mn)$ time processing the remaining deletions. Thus, if every individual deletion in a graph resulted in a new component, their algorithm would have a $O(m \log n)$ running time. There does not seem to be any immediate way to extend their technique to derive a fast algorithm for the batch deletion problem in which a batch of deletions results in a new component.

2.1 Algorithm A.

We can use a deletions-only dynamic connectivity algorithm for this problem. A *deletions-only dynamic connectivity algorithm* is a data structure that supports updates and answers queries on a graph with a fixed number of nodes, where an update is a deletion of an edge and a query is of the form: Are nodes i and j connected? The queries are answered in constant time. Adjustments to the data structure following each deletion can be done $O(\sqrt{n})$ time per edge deterministically [12, 18] or in $O(\log^3 n)$ expected time per edge, using the randomized algorithm of [20].

Theorem 1 *We can solve the batch deletion problem, in a graph in which all edges are eventually deleted, with the requirement that we discover all components after each batch, in $O(m\sqrt{n})$ time, or in $O(m \log^3 n)$ expected time, where $m = |E|$ and $n = |V|$.*

Proof. We begin by determining and labeling the initial components of G, and then, after a batch of edges are deleted from G, we delete all edges in that batch from our data structure one by one. After each edge (a, b) is deleted, we query the pair a, b. If a and b are no longer connected, we search from both a and b, alternating between the two searches, until one component is completely visited (this technique derives from [14]). We relabel the nodes of the completely visited component. The length of the search is thus proportional to the number of edges in the component with the fewer number of edges. Since each edge is in the component with the fewer edges at most $\log m$ times over the course of the algorithm, each edge is visited no more than $\log m$ times for a total cost of $O(m \log n)$ time.

If we use the best deterministic deletions-only dynamic algorithm for connectivity [12, 18], then the m queries and $O(m)$ updates cost $O(m\sqrt{n})$ time, while if we use the randomized fully dynamic algorithm of [20], we can perform all the queries and updates in $O(m \log^3 n)$ time. ∎

2.2 Algorithm B: A faster batch deletion algorithm.

In this section we will describe an algorithm which uses $O(n^2 \log n + b_0 min\{n^2, m \log n\})$ time for a variant of the batch deletion problem which we will now describe.

We will say that a component is *newly created by batch i* if it forms a component after batch i, but was part of a larger component after batch $i - 1$. The variant of the batch deletion problem we wish to solve will require only that we find at least one newly discovered component (see below for what this means), rather than all newly created components.

We continue with definitions.

The *size* of a component is the number of nodes in the component. Each node in a component has a pointer to the label of its component, and the label of a component consists of a pair $(name, size)$, where $name$ is a unique identificator and size is a positive integer. A component is *undiscovered* if its current label is shared by a component which had previously been connected to it. If after batch i the algorithm determines that the label of a component is no longer valid (since the size of the label does not agree with the actual size of the component) and after batch j with $j < i$ the algorithm had not discovered this, the component is *newly discovered after batch i*. Let C be any connected component of G which becomes disconnected after a batch of deletions. In the resulting graph, we call C' a *smaller component* if it is a connected component of C of size no greater than half the size of C.

Throughout the algorithm, for each undiscovered component which is found, we assign a label consisting of a unique name and its correct size.

Note that if there is an undiscovered component then there is also an undiscovered smaller component. By definition, there must be two or more undiscovered components with the same label, and all but one must be smaller. Similarly, if there is a newly created component, there is a newly created smaller component. The following algorithm finds an undiscovered smaller component after each batch of deletions. The algorithm may be modified, as explained below, to stop only when it finds an undiscovered smaller component which is also newly created. These modifications are parenthesized.

Batch deletion algorithm B:

- *Initialize:* Construct a sequence of graphs $G_1, G_2, ..., G_{\lg n} = G$ with $G_i = (V, E_i)$, and $E_i = \{(a,b) : min\{deg_G(a), deg_G(b)\} < 2^i\}$. For each node $v \in V$, if $deg_G(v) \geq 2^i$, then we color v blue in G_i, and otherwise we color v white. We label each node in each graph G_i with the size of its component in G_i. Note that an edge will in general appear in several graphs G_i.
- *Adjust each G_i:* Each time an edge e is deleted, we adjust the appropriate G_i by deleting e from all graphs G_i which contain e. If the degree of a node falls from 2^i to below 2^i, color the node white in G_i and add its remaining edges to G_i.
- *Find a (newly created) smaller undiscovered component:* After a batch is deleted, compute the connected components in each G_i starting with G_1 until a graph is found which contains a (newly created) undiscovered component C which contains only white nodes. Any connectivity algorithm which runs in time linear in the number of edges of the modified component may be used.

  A component is determined to be undiscovered and smaller by comparing its actual size to the size in the label of one of its nodes. (A component is determined to be newly created by checking the subset of edges most recently deleted which are incident to the component to see if one of the edges' endpoints lies in a different component.) Adjust the label of the component which previously contained C by subtracting out the size of this undiscovered component. Relabel the nodes in this undiscovered component.

  Output C.

Analysis of the Batch Deletion Algorithm. We start with the analysis of the smaller undiscovered component algorithm. The initialization and adjustments take $O(n^2)$ time since there are $\lg n$ graphs and the total number of edges in G_i is $O(2^i n)$.

We use the following observation:

Lemma 1 *If a set of nodes in G is a component of size no greater than 2^i then it appears as a component in G_i which contains no blue nodes.*

Theorem 2 *Algorithm B has running time $O(n^2 \log n + b_0 min\{n^2, m \log n\})$, where b_0 is the number of batches which do not result in the creation of a new component.*

Proof. A component of size no greater than 2^i will always be found in some G_j, for $j \leq i$, by Lemma 1. Since G_j has at most $O(2^j n)$ edges, it takes time $O(2^i n)$ to find a new component of size no greater than 2^i.

Over the course of the algorithm, for each i, a node can only be in one smaller component of size greater than 2^{i-1} and no greater than 2^i. Thus, there are no more than $n/2^{i-1}$ such components during the course of the algorithm. Thus the total cost of finding all smaller components of size greater than 2^{i-1} and no greater than 2^i during the course of the algorithm is $O(n/2^{i-1} * 2^i n) = O(n^2)$. Summing over all i, $1 \leq i \leq \log n$, the total cost of all searches which end in the discovery of a smaller component during the course of the algorithm is $O(n^2 \log n)$. Any search which does not end in the discovery of a smaller component takes $O(\min(n^2, m \lg n))$ time. ∎

Note that we can extend this analysis to the newly created option, by noting that the test to determine if a undiscovered component is newly created takes time proportional to the number of edges which were just deleted and which are incident to that component. Therefore, these tests involve looking at each edge at most twice during the algorithm. So the cost of these tests over the whole execution of the algorithm is proportional to m. The same analysis as above holds.

3 The subtree consistency problem

We begin with some definitions. Let T be a rooted binary tree leaf-labelled by a set S. For v, w nodes in T, we say that v *lies below* w if the path from v to the root of T passes through w. The *least common ancestor* of leaves a and b is the node v such that a and b are in the subtree of T rooted at v, and if a and b are also in the subtree of T rooted at w, then v lies below w.

Consider the following special case of the general subtree consistency problem we introduced in Section 1, in which every subtree has three leaves:

Rooted triple consistency:

Input: Set $\mathcal{T}$ of rooted triples $((a, b)c)$, indicating that the lca of a, b lies below the lca of a, c.

Output: Rooted tree T consistent with the rooted triples if such a tree exists.

We will show that each instance $T_1, T_2, \ldots, T_k$ of the subtree consistency problem can be encoded as an instance of the rooted triple consistency problem, using $O(N)$ rooted triples, where $N = \sum_i |V(T_i)|$.

The basic idea in the encoding is to include one rooted triple for each edge which is not incident to a leaf, in such a way that the set of rooted triples defined for the edges in the subtree T_v (i.e. the subtree rooted at the node v) will define T_v. Let e be an edge which is not incident to a leaf, and let v be its child. Let the right child of v be v_1 and let the left child be v_2. Let the rightmost leaf in the subtree rooted at v_1 be a, and let the rightmost leaf in the subtree rooted at v_2 be b. Let the parent of v be v', and let the rightmost leaf in the *other* subtree of v' be c. Then we will associate to the edge e the rooted triple $((a, b), c)$.

In this way, we can associate a set of rooted triples to each tree, with exactly one rooted triple per edge in the tree.

Theorem 3 *Let $T_1, T_2, \ldots, T_k$ be a set of rooted trees, let $\mathcal{T}_i$ denote the set of triples defined by T_i, for $i = 1, 2, \ldots, k$, and let $\mathcal{X} = \cup_i \mathcal{T}_i$. Then $\mathcal{X}$ is compatible with a tree T if and only if the trees $T_1, T_2, \ldots, T_k$ are compatible with T.*

The proof is straightforward and is omitted.

Consequently, we can design algorithms for the subtree consistency problem that are based upon rooted triple consistency.

3.1 Previous results

In 1981, Aho, Sagiv, Szymanski, and Ullman [3] addressed the problem of consistency of homeomorphic subtrees in a paper motivated by an application to relational databases. They presented an $O(Nn)$ algorithm to solve the case where each subtree is a rooted resolved (i.e. binary) tree on three leaves, where N is the number of rooted subtrees and n is the number of distinct labels on the leaves.

Given a set of rooted binary trees, $\mathcal{T}$, the algorithm in [3] constructs a graph $U_{assu} = (V, E)$ as follows. Let $V = \{1, 2, \ldots, n\}$, and $E = \{(p, q) : \exists((p, q), r) \in \mathcal{T}\}$. The connected components of U are computed using any linear time algorithm. If U is connected, then there is no consensus tree; otherwise, let $G_1, G_2, \ldots, G_k$ be the distinct components of U_{assu}. For each G_i, and for each tree $T \in \mathcal{T}$, determine if T applies to G_i (T applies to G_i if all of the leaves in T are in $V(G_i)$), and recurse on each subproblem $(G_i, \mathcal{T}^i)$, where $\mathcal{T}^i$ is the set of trees in $\mathcal{T}$ which apply to G_i. A consensus tree exists if and only if each subproblem has a solution. Let $T'_1, .., T'_k$ be the consensus trees generated for the k subproblems. Then the consensus tree for the input is given by adding edges from the roots of the T'_i to a new parent node. The proof of correctness of this algorithm may be found in [3].

Note that, as stated, the algorithm may require $\Theta(nN)$ time since at each stage, it has to examine each $T \in \mathcal{T}$, and determine whether T "applies" to G_i (when T does not apply to G_i, the edge associated to T is then deleted from G_i, so that the graphs modify via edge-deletions). *After* this step is accomplished, the algorithm then recomputes connected components. Note that the first step (determining whether each $T \in \mathcal{T}$ applies to each G_i) itself takes $O(|\mathcal{T}|)$ time, and thus, over the course of the algorithm, the contribution to the running time of just this part of the algorithm is $O(Nn)$ time. The second step, recomputing the components of each G_i, also takes $O(m)$ time per iteration of the algorithm (where m is the number of edges in the graph U_{assu}, so $m \leq min\{n^2, |\mathcal{T}|\}$), and hence contributes $O(n^3)$ to the runtime. Since N can be $\Omega(n^3)$, this is an $O(n^4)$ algorithm.

Note that there are two bottlenecks to this approach: the determination of the edges to be deleted from the graph U_{assu}, and the recomputation of the components of U_{assu} after each set of edges is deleted.

3.2 Faster algorithms for rooted triple consistency.

We use the same basic ideas as the Aho *et al.* algorithm in that we compute components of the graph U_{assu}, obtain solutions to subproblems, and combine them appropriately. However, we obtain a faster runtime than the Aho *et al.* algorithm by avoiding the two bottlenecks described above. First, as we have noted, the graph U_{assu} changes by having edges deleted; thus, we can use our algorithms for the batch deletion problem here to speed up the computation of the new components of the graph (and hence to reduce to subproblems). We also speed up the determination of the edges which need to be deleted. Rather than explicitly examining each rooted subtree T to see if it still applies, we use an auxiliary data structure (which we describe below) to quickly determine exactly those subtrees T which no longer apply to the problem formulation. A coordinated use use of the two datastructures (the batch deletion data structure and our auxiliary data structure) then provides the desired speedup.

Although we generally follow the Aho *et al.* algorithm, we allow our method to either find *all* components, or only at least one.

If we use the version of the batch deletion algorithm which discovers *all* newly created components, the consensus tree with the minimal number of nodes (the *minimal tree*) is returned. When the batch deletion algorithm only determines a newly discovered component, then a binary tree is returned which will be consistent with the constraints but not necessarily minimal. Depending upon the particular application, the minimality may or may not be important (it turns out to be important for the database problem, but not necessarily important for the biological problem).

3.3 The data structures.

We now define our data structures.

Data Structures: a directed graph D and an undirected graph, U.

- $U = (V, E)$ with the vertex set $V = \{1, 2, ..., n\}$ and where, for each $((a, b), c) \in \mathcal{T}$, $(a, b) \in E$.

- $D = (V', A)$ where for each $((a,b),c) \in \mathcal{T}$ nodes (a,b) and (b,c) are in V' and $(a,b) \to (b,c)$ is in A.

Call a node in D which has outdegree 0 a *terminal* node, and all other nodes *non-terminal.* Note that there is initially a 1-1 correspondence between the edges in U and the non-terminal nodes in D; however, during the course of the algorithm, this correspondence may no longer apply.

3.4 Structure of our algorithms.

We define some terminology which will be used in the description of the algorithm. The *yellow components* of U are the components of the subgraph of U defined by the yellow edges of U. A red edge in U whose endpoints are in different yellow components is called a *separable red edge.*

The structure of each of our algorithms is as follows:

Step 1: Construct U and D. Color all nodes in D and edges in U *yellow.* Compute yellow components in U. (Note that $U = U_{assu}$ at this point.)

Step 2: Identify terminal nodes in D. Recolor these nodes red. If their corresponding edges exist in U, color these red as well (in the first iteration, there are no edges of U which correspond to terminal nodes in D, but in subsequent iterations there may be).

Step 3: If U has no edges, then return the consensus tree T with a root and all nodes in U children of the root. Otherwise, recompute yellow components of U using one of the algorithms presented in Section 2. If no new component is found, then stop and return *Null-tree*, else let $C_1, C_2, \ldots, C_k$ be the new component(s) found. Make a tree T with root r; the subtrees of T will be the trees constructed on each of the components $C_1, C_2, \ldots, C_k$. For each new component found, identify the set E_{sep} of separable red edges incident to that new component. Delete these edges from U and the corresponding nodes from D. Go to Step 2.

It is clear that the determination of the set of separable red edges is the singly most important implementation detail we need to establish. We will modify the batch deletion algorithms of Section 2 to find the separable red edges.

3.5 Analysis of running time.

The construction of U and D takes time proportional to $N = \sum_i |T_i|$. The overall cost of determining the maximal nodes of D is proportional to the size of D, i.e. $O(N)$. We use one of the batch deletion algorithms to find a new component in each phase. While discovering a new component C, we also examine and identify the red edges incident to it. Each separable red edge e is then discarded from U, for a total cost of $O(N)$. The cost of using batch deletion algorithms to compute the consensus tree then depends upon the cost of the batch deletion algorithm itself, and the number of times non-separable edges are visited.

We thus have two different consensus tree algorithms, depending upon which of the two batch deletion algorithms we use. We will refer to these two consensus tree algorithms as **Algorithm A'** and **Algorithm B'**, corresponding to **Algorithm A** and **Algorithm B** respectively. In each case, the use of the batch deletion algorithm is straightforward, and all we need to discuss is how we identify the separable red edges.

3.6 Algorithm A'.

Algorithm A' uses Algorithm A to recompute connectivity and determine all components after each batch of edge deletions. When we delete a batch of edges, for each edge (a, b) deleted such that a and b are now in separate yellow components, we search from the smaller of the two components found for all red edges that are incident to vertices in that new component. Each separable red edge that is found is deleted from the graph.

Theorem 4 *Using Algorithm A', we can construct the minimal consensus tree of N rooted triples on n vertices in $O(N\sqrt{n})$ time, or in $O(N \log^3 n)$ expected time.*

Proof. It is clear that the tree constructed by Algorithm A' is identical to the tree constructed by the algorithm in [3], and hence it is minimal. For the same reason, Algorithm A' determines correctly whether the set of triples is consistent. Thus, it remains only for us to show that the running time is as stated.

The batch deletion algorithm (Algorithm A) has a deterministic version which has running time $O(m\sqrt{n})$ and a randomized version which has $O(m \log^3 n)$ expected time. We will show that the additional cost of finding all separable edges during each iteration will cost only an additional $O(m \log n)$ time, so that we remain within the bounds as stated.

Recall that the algorithm searches for separable red edges by exploring the smaller of the two components C_a and C_b, where $a \in C_a$ and $b \in C_b$, and (a, b) is an edge deleted in the most recent batch such that a and b are now in separate components. This search discovers separable red edges, which are then deleted (and hence never explored again), for a total contribution $O(m)$ to the cost of the algorithm. Each non-separable edge is visited at most $\log n$ times, for a total contribution of $O(m \log n)$ to the cost of the algorithm. ∎

The deterministic algorithm we have just given is faster than the Aho et al algorithm; however, we will show that we can use Algorithm B, the faster batch deletion algorithm, to create an even faster deterministic algorithm for subtree consistency. We will call this Algorithm B'.

3.7 A faster algorithm for consensus tree construction.

Once again, we need to show how we explicitly find the separable red edges, since otherwise the use of Algorithm B to recompute connectivity is straightforward.

In each iteration, Algorithm B either finds a new smaller component or fails to do so and returns a null-tree indicating failure. If Algorithm B finds a new smaller component C, we then explore all red edges incident to C; the separable red edges are then deleted from U.

Theorem 5 *Algorithm B' returns a tree T for input $\mathcal{T}$ if and only if T is a consensus tree for $\mathcal{T}$.*

Proof. Suppose a non-null tree T is returned. Let $((a, b), c) \in \mathcal{T}$ be one of the input triples. Then D has the arc $(a, b) \to (b, c)$ and U has edges (a, b) and (b, c). Since T is returned, at some point these edges are deleted, and thus the nodes in U corresponding to (a, b) and (b, c) are turned red and deleted. When the algorithm removes node (b, c) from D, it is because we have discovered a new smaller component C, and exactly one endpoint of (b, c) is in C. Without loss of generality, let us say that endpoint is c. The algorithm then constructs a tree T_1 for C, and another tree T_2 for $C' - C$ (where C' was the component originally containing b and c both, before the previous batch deletion discovered C). Since a and b are still connected by a yellow edge at this point, the topology of the tree T on a, b, c has the form $((a, b), c)$, as required. Thus, if a tree T is returned, it is a consensus tree for $\mathcal{T}$.

Now suppose a consensus tree exists, but that this algorithm returns a null-tree. If this happens, then there is some set C of nodes in U such that the graph (C, E_C) is connected, where E_C is defined by $\{(x,y) : \exists((x,y),z) \in \mathcal{T}, \{x,y,z\} \subseteq C\}$. But then the Aho *et al.* algorithm would also fail to produce a tree, and hence no consensus tree exists. ∎

Theorem 6 *Using Algorithm B', we can determine consistency of N rooted subtrees on n leaves and construct the consensus tree if it exists, in $O(N + n^2 \log n)$ time.*

Proof. By Theorem 5, the algorithm returns a non-null tree if and only the input is consistent.

We now show that when using Algorithm B for batch deletions, we have $b_0 \leq 1$, where b_0 is the number of batches that do not result in a new component. Since the algorithm terminates if any batch does not result in a new component, this is trivial. The remaining work, as we have shown above, is therefore in finding all non-separable edges. Each non-separable edge e found goes between vertices in the same component. Since each new component found is a *smaller* component, each node is in a smaller component at most $\lg n$ times and therefore each edge is considered at most $2 \lg n$ times. Thus, the total cost to us for considering the non-separable edges is $O(n^2 \log n)$. Thus, the total cost of this algorithm is $O(N + n^2 \log n)$. ∎

4 Application to Evolutionary Tree Construction

An evolutionary tree for a species set S is a rooted tree T with the leaves bijectively labelled by the species in S. Constructing evolutionary trees for biological species is a difficult problem for a variety of reasons, which we will now briefly summarize.

Current methods for constructing evolutionary trees are typically based upon attempts to solve certain optimization problems, almost all of which are NP-hard or conjectured to be NP-hard [9, 10, 15, 2]). More problematic than the intractability of the optimization problems used to reconstruct trees is the observation that very large data sets may simply be hard to analyze using existing approaches, even if the underlying optimization problems were to be solved exactly (see [13, 29, 5]).

One proposal [25] for handling these difficulties is to separate the data set into overlapping subsets, each of which is amenable to analysis, and then to combine the subtrees that result into one supertree; such an analysis has yielded interesting results on some data sets [11]. It is clear that the problem this paper addresses fits appropriately into this framework, so that the algorithms we have provided can be used directly to infer trees from subtrees constructed on subproblems. Another approach requires that all the sequences be used in one data analysis, rather than separated; this is called the *total evidence* approach. In this case, there are likely to be many disparate trees which then need to be analyzed in order to gain some understanding of the evolutionary tree. This suggests that *consensus* approaches are needed, in which one tree is constructed from a profile of trees. There have been many suggestions for how to infer the *consensus tree* from a profile of trees (see [1, 4, 6, 7, 8, 16, 19, 24, 23, 26, 27, 28, 30, 31] for some of the literature on this rich subject). One of these methods is the *Local Consensus Tree* [23], which is based upon combining information on subtrees from each of the trees in the profile. We will show that our algorithm for subtree consistency can be used to solve the Local Consensus Tree problem faster than could be solved before.

A *local consensus rule* determines the exact form of (possibly) each rooted subtree of every triple of species drawn from S, based upon the form of each of the trees in the input profile for that triple. A *local consensus tree* is then required to have the form specified by the local consensus rule for each triple a, b, c of species (unless the local consensus rule does not specify the form on that triple).

For example, a local consensus rule may require that if all the trees in the profile have the same form on a triple a, b, c, then the output tree T must have that form as well, and otherwise the output tree must be *unresolved* (i.e. the subtree of T defined on a, b, c must be a star). Given such a local consensus rule and a profile of rooted trees, the objective is to determine whether there is a tree that meets all the constraints. It is easy to see that for some local consensus rule and profiles of rooted trees, there may be no such tree.

Some local consensus rules, such as the one just described, specify the form of each rooted triple in the output tree; these are said to be *entire*. When the local consensus rule is *entire*, it is possible to construct the local consensus tree by using the algorithms of [21]. In the case where the local consensus rule is not entire, so that the form of the output tree on some triples is not constrained, then we can instead use one of the subtree consistency algorithms we have described in this paper, provided that all the constraints require the output to be *resolved* subtrees (i.e. binary). If some of the constraints enforce star topologies on triples of leaves, then we may use a variant of the Aho *et al.* algorithm instead (see [3]), which allows subtrees to be unresolved.

It is worth noting that biologically relevant local consensus rule will not, in general, require that the output tree be unresolved (i.e. be required to have a star topology) on a triple. This means that the constraints indicated by a biologically relevant local consensus rule are equivalent to testing consistency of rooted binary trees.

Our algorithm is as follows.

Step 1: We apply the local consensus rule to each triple, a, b, c, of species; for each triple such that the form of the consensus tree is determined, we include the rooted tree on a, b, c in the set $\mathcal{T}$.

Step 2: We now pass $\mathcal{T}$ to a consistency algorithm, and determine if a tree consistent with each of these rooted triples exists, and construct it if it does.

Theorem 7 *We can construct the local consensus tree of k trees (assuming all returned triples are resolved, and so are binary trees) in $O(kn^3)$ time.*

Proof. Step 1 takes $O(kn^3)$ time. This returns a subset of $m \leq n^3$ rooted triples, which can then be passed to a subtree consistency algorithm. The determination of subtree consistency then takes $O(min\{m + n^2 \log n, m\sqrt{n}\})$. Since $m \in O(n^3)$, the cost of consistency of m subtrees is bounded from above by $O(n^3)$; hence the total cost is bounded by $O(kn^3)$. ∎

Note that by contrast, if we use the Aho et al algorithm for the subtree consistency, this will result in an $O(kn^3 + n^4)$ algorithm.

For general information of evolutionary tree construction problems and methods, and for further details about some of the issues we raised above, see [17].

References

[1] E. Adams III, *Consensus techniques and the comparison of taxonomic trees,* Systematic Zoology 21 (1972), pp. 390-397.

[2] R. Agarwala, V. Bafna, M. Farach, B. Narayanan, M. Paterson, and M. Thorup, *On the approximability of numerical taxonomy: fitting distances by tree metrics.* Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, 1996.

[3] A.V. Aho, Y. Sagiv, T.G. Szymanski, and J.D. Ullman. *Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions*, SIAM J. of Computing, Vol 10, No. 3, 1981, pp. 405-421.

[4] A. Amir and D. Keselman, *Maximum agreement subtree in a set of evolutionary trees – metrics and efficient algorithms*, to appear, SIAM Journal on Computing.

[5] K. Atteson, *The performance of neighbor-joining algorithms of phylogeny construction,* Proceedings, 1997 Computing and Combinatorics (COCOON), Springer-Verlag Lecture Notes in Computer Science, Vol. 1276, pp. 101-110.

[6] J.P. Barthelemy, and F.R. McMorris, *The median procedure for n-trees*, Journal of Classification 3, pp. 329-334, 1986.

[7] D. Bryant, *Building Trees, Hunting for Trees, and Comparing Trees, (Theory and methods in phylogenetic analysis)*, 1997, Ph.D. Thesis, Department of Mathematics, Canterbury University, Christchurch New Zealand.

[8] W.H.E. Day, *Optimal algorithms for comparing trees with labelled leaves*, Journal of Classification 2, pp. 7-28, 1985.

[9] W.H.E. Day, D.S. Johnson, and D. Sankoff, *The computational complexity of inferring rooted phylogenies by parsimony,* Mathematical Biosciences, 81:33-42, 1986.

[10] W.H.E. Day, and D. Sankoff, *Computational complexity of inferring phylogenies by compatibility,* Syst. Zoology, 35, 224, 1986.

[11] M.J. Donoghue, *Phylogenies and the analysis of evolutionary sequences, with examples from seed plants.* Evolution 43 (6): 1137-1156, 1989.

[12] D. Eppstein, Z. Galil, and G. F. Italiano. *Improved Sparsification*, Tech. Report 93-20, Department of Information and Computer Science, University of California, Irvine, CA 92717.

[13] P. Erdos, M. Steel, L. Szekely, and T. Warnow, *A few logs suffice to build almost all trees*, Proceedings ICALP 1997.

[14] S. Even and Y. Shiloach, *An on-line edge deletion problem,* J. ACM, Vol. 28, No. 1 (1981), pp. 1-4.

[15] M. Farach, S. Kannan and T. Warnow, *A Robust Model for Finding Optimal Evolutionary Trees*, Algorithmica, special issue on Computational Biology, Vol. 13, No. 1, 1995, pp. 155-179.

[16] M. Farach, T. Przytycka, and M. Thorup, *On the agreement of many trees*, Information Processing Letters, to appear.

[17] J. Felsenstein, *Numerical methods for inferring evolutionary trees,* The Quarterly Review of Biology, Vol. 57, No. 4, (1982), pp. 379-404.

[18] G. N. Frederickson, *Data Structures for On-line Updating of Minimum Spanning Trees* SIAM J. Comput. 14:781 – 798, 1985.

[19] D. Gusfield, *Efficient algorithms for inferring evolutionary trees*, Networks 21, pp. 19-28, 1991.

[20] M. Henzinger and V. King, *Randomized Dynamic Algorithms with Polylogarithmic Time per Operation*, Proceedings, STOC '95.

[21] S. Kannan, E. Lawler, and T. Warnow, *Determining the Evolutionary Tree*, Journal of Algorithms, 1996, 21(1): 26-50.

[22] S. Kannan and T. Warnow, *A fast algorithm for the computation and enumeration of perfect phylogenies when the number of character states is fixed*, SIAM Journal on Computing, December 1997.

[23] S. Kannan, T. Warnow, and S. Yooseph, *Computing the local consensus of trees*, Proceedings, SODA '95. Also, to appear, SIAM Journal on Computing.

[24] M.Y. Kao, *Tree contractions and evolutionary trees,* SIAM Journal on Computing, to appear.

[25] M. Miyamoto and W. Fitch, *Testing species phylogenies and phylogenetic methods with congruence,* Systematic Biology 44 (1995), 64-76.

[26] G. Nelson, *Cladistic analysis and synthesis: Principles and definitions, with a historical note on Adanson's Familles des Plantes (1763-1764)*, Syst. Zoology 28 (1979), 1-21.

[27] R.D.M. Page, *Genes, Organisms and Areas: The problem of multiple lineages*, Systematic Biology 42(1), pp. 77-84, 1993.

[28] C. Phillips and T. Warnow, *The asymmetric median tree: a new model for building consensus trees*, Discrete Applied Mathematics, 1997.

[29] K. Rice and T. Warnow, *Parsimony is Hard to Beat!*, Proceedings, Computing and Combinatorics (COCOON) 1997, Springer-Verlag Lecture Notes in Computer Science, pp. 124-133.

[30] M. Steel and T. Warnow, *Kaikoura Tree Theorems: Computing the maximum agreement subtree*, Information Processing Letters 48, pp. 72-82, 1993.

[31] T. Warnow, *Tree Compatibility and Inferring Evolutionary History,* Journal of Algorithms, 16, pp. 388-407, 1994.